Problems with using MPI 1.1 and 2.0 as compilation targets for parallel language implementations

Dan Bonachea and Jason Duell
Computer Science Division
University of California, Berkeley
Berkeley, California, USA

E-mail: bonachea@cs.berkeley.edu, jcduell@lbl.gov

Keywords: MPI one-sided, parallel languages, global address space, RMA, one-sided communication.

Abstract

MPI support is nearly ubiquitous on high performance systems today, and is generally highly tuned for performance. It would thus seem to offer a convenient "portable network assembly language" to developers of parallel programming languages who wish to target different network architectures. Unfortunately, neither the traditional MPI 1.1 API, nor the newer MPI 2.0 extensions for one-sided communication provide an adequate compilation target for global address space languages, and this is likely to be the case for many other parallel languages as well. Simulating one-sided communication under the MPI 1.1 API is too expensive, while the MPI 2.0 one-sided API imposes a number of significant restrictions on memory access patterns that that would need to be incorporated at the language level, as a compiler can not effectively hide them given current conflict and alias detection algorithms.

1 Introduction

Global address-space (GAS) languages, such as Unified Parallel C (UPC) [10], Titanium [30], and Co-Array Fortran [26], are an emerging class of languages that seek to give parallel application developers an alternative to the traditional message passing model. By providing the illusion of a shared address space to a distributed program (regardless of the underlying hardware), along with programmatic awareness and control of memory layout across processors, they promise to combine the performance of message passing systems with the convenience of shared memory programming.

As researchers involved in developing portable implementations of two of these languages (Berkeley UPC [7] and Titanium [30]), we have invested (and continue to invest) substantial amounts of effort into porting our software to new network architectures and programming interfaces.

Many other scientists and hardware vendors, upon learning this, have asked us why we do not simply write our language on top of MPI, to avoid this porting effort. As the most common parallel programming interface on contemporary supercomputing platforms, MPI has typically been highly tuned for performance. It has also been described by some of its developers as providing an "assembly language for parallel processing" [13]. Presumably, then, it ought to provide an efficient, portable target for parallel language compilers.

Unfortunately for GAS language implementors, this is not the case. This paper explains why both the traditional MPI 1.1 two-sided interface [25] and the newer MPI 2.0 [24] one-sided interface (hereafter referred to as MPI-RMA) are inadequate compilation targets for global address space languages. The two-sided communication model inherent in MPI 1.1 cannot support the one-sided communication requirements of GAS languages as efficiently as lower-level APIs. The newer one-sided MPI-RMA presents an interface that, while undoubtedly of great use to many application developers, presents a number of restrictions on memory access patterns that would require conflict and alias analysis that is beyond the reach of current compilers. The difficulties presented by MPI-RMA as a compilation target are not specific to GAS languages, and are likely to present an obstacle to any parallel programming language which does not expose application developers to MPI-RMA's numerous restrictions on memory usage patterns.

1.1 Global Address Space languages and their communication requirements

The common characteristic of all GAS languages is that they provide programmers with a shared memory abstraction between all processors in an application (regardless of whether shared memory is actually supported natively in hardware), and that the programmer is given both knowledge and control of the layout of shared memory across processors. The shared memory model of GAS languages provides the programmer with a familiar interface in which any piece of shared memory can be accessed by any processor in an application via the normal data access mechanisms built into the language, i.e., by accessing an element of a shared array, dereferencing a pointer to shared data, or simply referencing the name of a shared variable. Of course, on many platforms the access time for touching shared memory will depend heavily on the location of the data, and so GAS languages make the layout of shared memory explicit to the programmer, who is encouraged to write code that takes advantage of locality in order to achieve higher performance.

This combination of a globally shared address space with locality information allows an incremental development model, in which serial or shared-memory codes can initially be naïvely ported to a distributed environment, profiled to isolate bottlenecks, and then gradually tuned to optimize performance. The compiler for a GAS language may also be able to hide some or all of the latency cost of remote accesses by scheduling unrelated computation (or more communication) during the interim imposed by network traffic.

GAS languages have some differences in how they allow shared memory to be allocated, and/or how they distinguish shared and non-shared data. UPC and Co-Array Fortran require the programmer to explicitly declare data objects to be either shared or private (usage varies by application, but typically the major data structures reside in shared space). In Titanium, by contrast, all heap and static data can potentially be accessed remotely [14], although a sophisticated compiler escape analysis [19] can detect which objects are potentially "shared" (interesting applications typically have about 50-100% of the total bytes allocated judged to be shared by the analysis). Dynamic allocations of shared memory must be collective in Co-Array Fortran, while they can also be achieved in UPC and Titanium through purely local, non-collective operations. UPC also allows shared objects to be allocated on remote processes using a non-collective operation (upc_global_alloc()) with no explicit cooperation from the process allocating the data.

Remote data access in GAS languages naturally has a one-sided communication pattern. In part, this is a matter of providing a familiar and convenient interface to programmers who are used to a shared-memory programming model, but there is also an important class of dynamic and irregular applications that we wish to support which have communication patterns that are data-dependent and therefore statically unpredictable, and hence are most naturally expressed using one-sided operations.

In GAS languages, the ability to predict data access patterns is further limited by the fact that all three of these GAS languages allow accesses to shared objects with affinity to the calling thread via "local" pointers, and these accesses are indistinguishable from accesses to purely local (private) objects, i.e., the languages' semantics provide no explicit information about whether the memory being accessed is potentially shared or not. In each language, accesses to shared data which reside locally using "local" pointers generally provide significantly better performance than access through "global" pointers. In fact, the performance impact is so dramatic that most UPC programmers specifically optimize for this case, and the Titanium compiler includes a specialized analysis [18] to automatically infer when such a transformation is provably legal.

A result of these data dependent access patterns and shared/local aliasing is that compilers for GAS languages generally have no way to know a priori which specific shared memory locations will be accessed remotely, or when they will be accessed. They thus have no way to statically check, for instance, if data accesses from different processors will conflict at runtime. This is exactly analogous to the problem of static alias analysis in sequential, pointer-based languages, but is complicated by the existence of multiple independent threads of control – for in-depth discussion, see [16, 27, 23].

Given these properties of GAS languages, they present the following requirements for any underlying network interface:

- The ability to perform (or at least simulate) one-sided communication (i.e., where only the initiator is explicitly involved and the communication proceeds independently of any action taken by the remote threads). While one-sided communication can be simulated by the language runtime using an underlying two-sided communication API, neither the user nor the compiler may be exposed to any aspect of implementing message receipts (or other bookkeeping) on the remote side of an access.

- Good latency performance for small remote accesses. Small messages are commonly used in initial implementations of many GAS programs, and may be unavoidable even in well-tuned applications in certain problem domains. The performance of applications written in global-address space languages is thus often very sensitive to network latency.

- The ability for the compiler to hide network latencies by overlapping communication with computation and/or other communication, through the use of non-blocking remote accesses. This requires that the software overhead involved in network traffic be less than the latency of messages (and/or the gap between how often the network interface will accept overlapping messages), as overlapping is impossible if the host CPU is busy throughout a messaging operation.

- Support for arbitrary access patterns to shared data. This includes allowing concurrent remote accesses to the same regions of shared memory from different remote processors, as well as support for allowing shared data to be accessed at arbitrary points in the program via local pointers.

- Support for using or implementing collective communication and synchronization operations.

The rest of this paper shows that neither the MPI 1.1 or the newer MPI-RMA interfaces support all of these characteristics adequately.

2 MPI 1.1 as a compilation target for GAS languages

The most widely available and portable software interface for programming distributed memory machines today is that provided by the MPI 1.1 specification, which has been implemented and carefully tuned on most contemporary high-performance parallel systems. As such, it presents a very tempting compilation target for GAS languages, which could potentially run on all such systems simply by providing a single networking layer that runs on top of MPI.

Communication under MPI 1.1 is strictly two-sided: all traffic takes the form of message sends, which require matching receive operations to be explicitly issued on the receiving side. While this does not neatly match the GAS model of one-sided communication, it does not disqualify MPI 1.1 from use by GAS languages. Our group has implemented an MPI layer [1] which transparently handles message receipt in the runtime by periodically polling for new message receipts, thus providing the illusion of a one-sided interface to both the user and the compiler. We have used this MPI layer on a number of platforms, including the IBM SP, the Cray T3E, the SGI Origin 2000, Linux and Compaq Alphaserver systems using Quadrics network hardware, and Linux clusters using Myrinet, Dolphin/SCI, and Ethernet networks. This portable MPI layer has proven to be an invaluable tool for quickly deploying our systems on new architectures.

However, as Figure 1 shows, use of MPI 1.1 implementations does not come without a cost. This microbenchmark data gathered by our group (and described in detail in [4]), demonstrate that the latency and/or software overhead associated with small message sends in MPI is typically much higher than when using native network APIs. This difference is most pronounced on the lowest-latency systems which are the most likely targets for GAS applications, with more than a factor of five difference between MPI performance and that of some native machine APIs.

It should be noted that the MPI data in Figure 1 were gathered using nonblocking send and receive calls, and that these calls tend to incur higher overhead than their blocking counterparts in most MPI implementations [4]. It is not clear whether these nonblocking overheads are unavoidable, or are simply the result of less tuning by vendors: most MPI applications use blocking functions, and vendor benchmarks generally report only numbers for blocking send/recv calls. Use of at least some nonblocking calls is necessary when simulating one-sided communication with a two-sided API, in order to prevent deadlock (which could otherwise be caused trivially by two processors sending a message to each other at the same time). Nonblocking calls are also desirable in a GAS language context, as compilers may be able to overlap computation (and/or other network calls) within the interval where the CPU would otherwise be blocked.

To more easily support porting our UPC and Titanium implementations, we use a portable, high-performance networking library called GASNet [5]. This is the layer which provides the one-sided messaging abstraction over MPI 1.1. Besides MPI 1.1, GASNet has also been implemented directly over the native APIs of a number of high-performance networks (the IBM SP's LAPI [17], Myrinet's GM [12], Quadrics' elan [11], and Infiniband [15]). UPC and Titanium applications can switch the underlying network used with a simple recompilation, and this gives us the ability to compare the performance of a single application run using MPI 1.1 for communication with its timing when run over a lower-level network API. Figures 2, 3, and 5 show the difference in performance between a set of UPC applications running on an Compaq Alphaserver system with a Quadrics interconnect: a naïve and a bulk-synchronous version of the NAS Parallel Benchmarks [2] Conjugate Gradient, and a bulk-synchronous implementation of the NAS Multigrid benchmark. Figures 4 and 6 show the same bulk CG and MG codes running on an x86-Linux Myrinet cluster. Finally, Figure 7 shows the performance of the MG benchmark on an IBM SP Power3. The applications were compiled with the Berkeley UPC Compiler v1.0beta [7] to use either MPI-1.1 or the lower-level elan layer for communication. For comparison purposes, on the Compaq system we also show the performance for the same applications when compiled with the 2.1-003 version of the Compaq UPC compiler [8], which compiles executables to use the elan API (data for Compaq UPC is omitted for higher numbers of nodes due to a recently-discovered performance bug which is still pending). In all cases, the network and system hardware being used is identical–the only difference is the

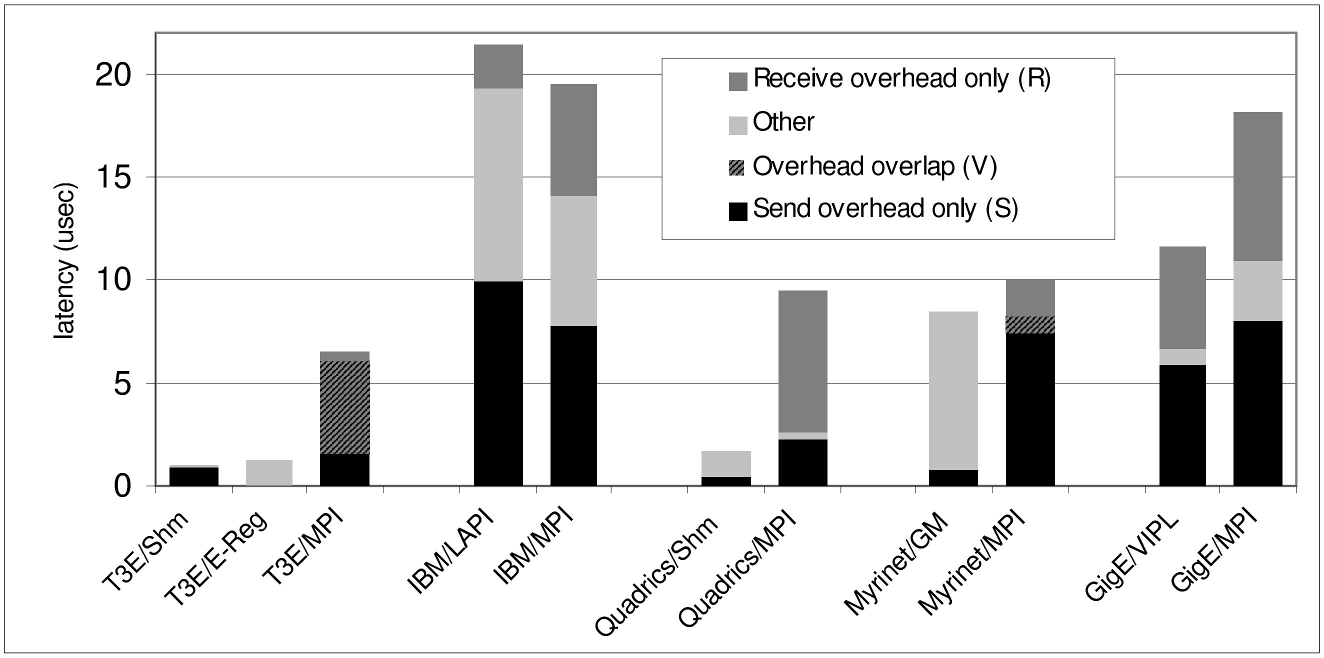

Figure 1. Send and receive software overheads (o_s and o_r) superimposed on the one-way end-to-end latency (EEL) for 8-byte messages on various high-performance systems. For MPI on the T3E and Myrinet, the sum of the overheads is greater then EEL, and so $o_s = S + V$ and $o_r = R + V$. For the other configurations $o_s = S$ and $o_r = R$.

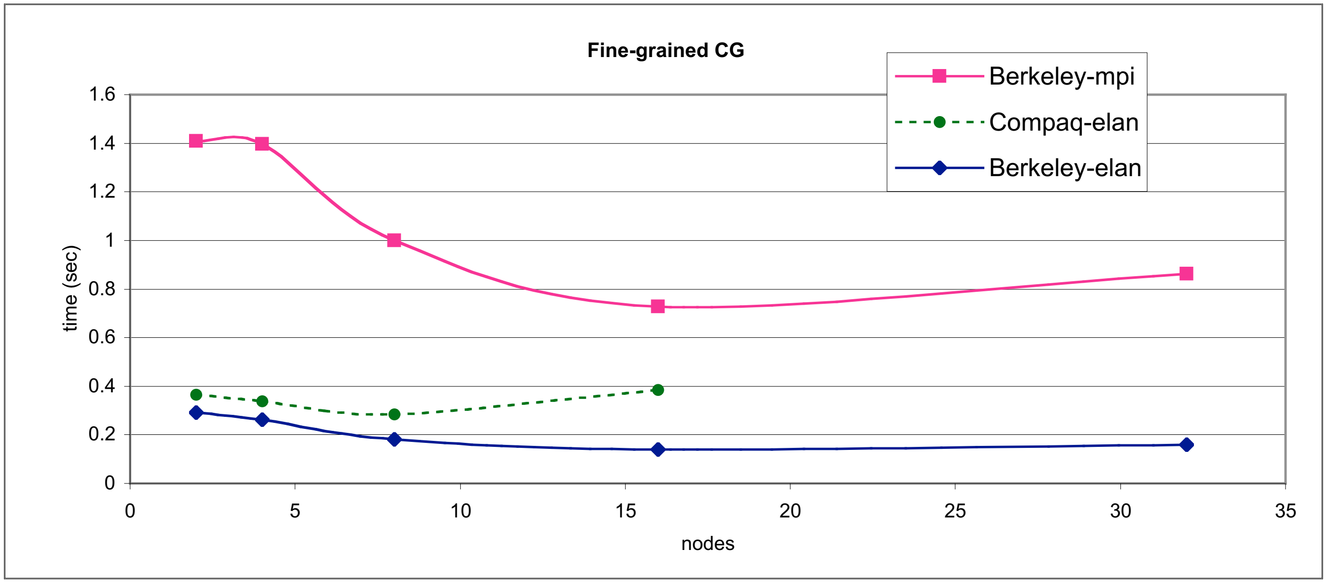

Figure 2. Performance for a naïve (fine-grained) UPC implementation of NAS Conjugate Gradient using different network APIs/compilers on a Compaq AlphaServer

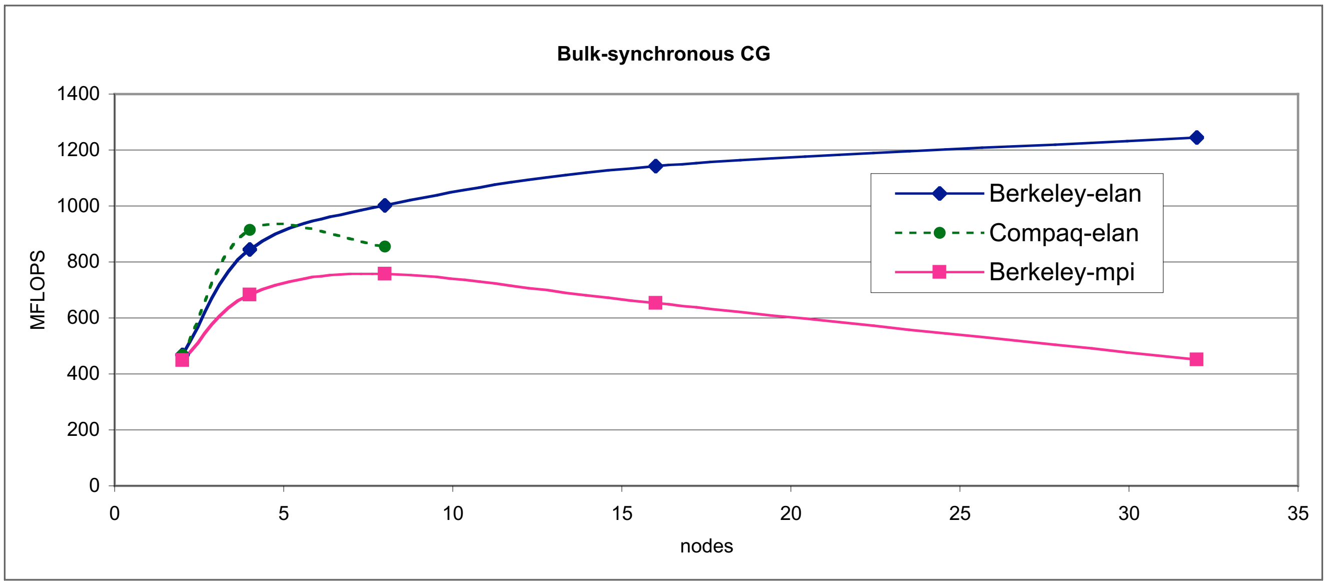

Figure 3. Performance for a bulk-synchronous UPC implementation of NAS Conjugate Gradient using different network APIs/compilers on a Compaq AlphaServer

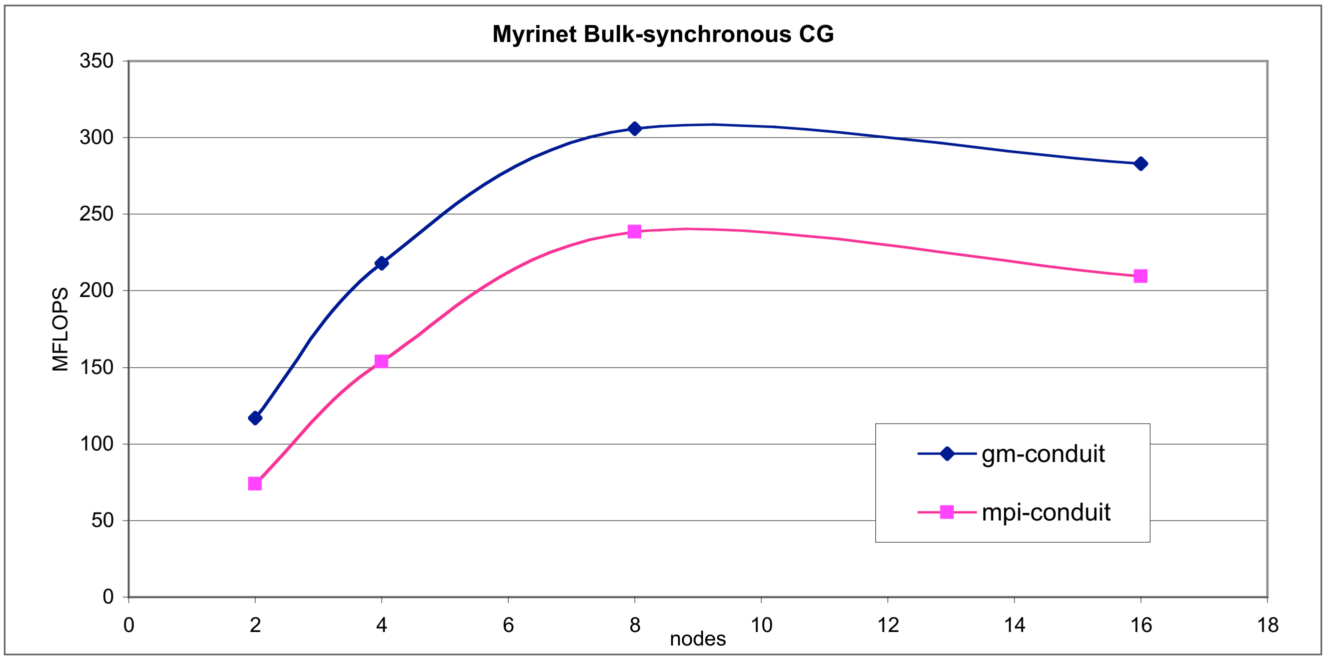

Figure 4. Performance for a bulk-synchronous UPC implementation of NAS Conjugate Gradient using different network APIs with the Berkeley UPC compiler over Myrinet/GM

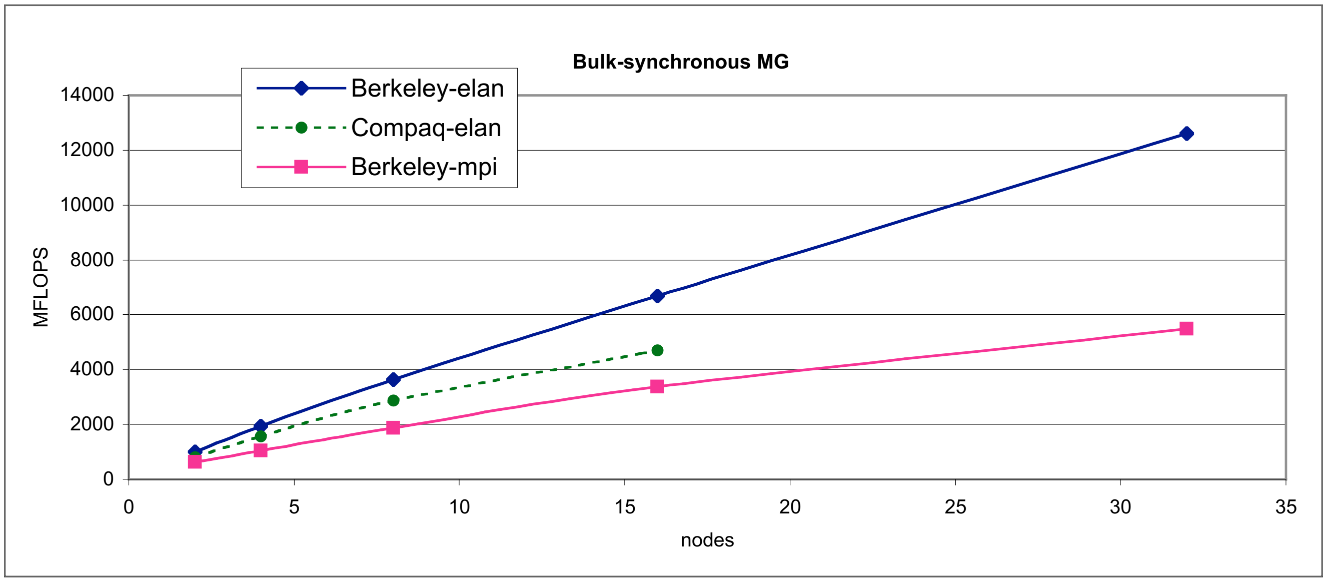

Figure 5. Performance for a bulk-synchronous UPC implementation of NAS Multigrid using different networks APIs/compilers on a Compaq AlphaServer

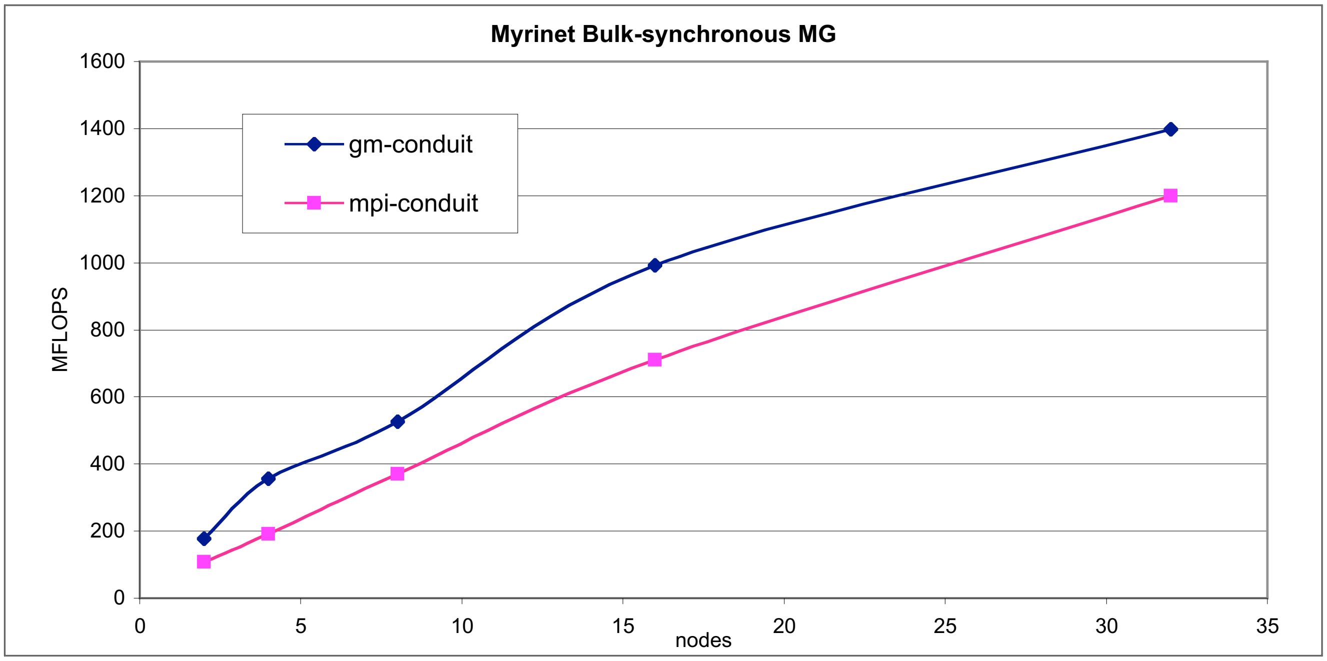

Figure 6. Performance for a bulk-synchronous UPC implementation of NAS Multigrid using different network APIs with the Berkeley UPC compiler over Myrinet/GM

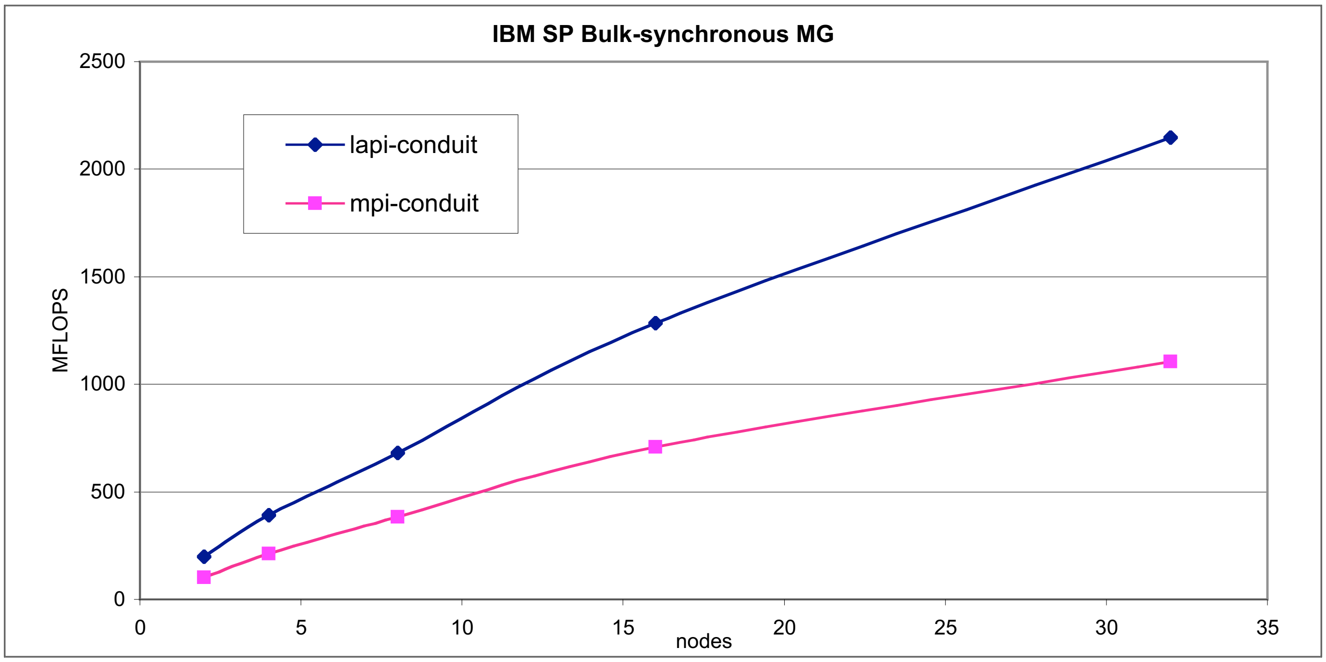

Figure 7. Performance for a bulk-synchronous UPC implementation of NAS Multigrid using different network APIs with the Berkeley UPC compiler on an IBM SP

runtime system and communication software in use.

The results show that the MPI 1.1-based GASNet layer is significantly outperformed by those based directly over one-sided networking APIs. The communication costs of the bulk-synchronous CG code are dominated by bulk put operations (avg 56KB each) and some small (8 byte) gets, whereas the MG code is dominated by bulk get operations (avg 100KB each) and some small (8 byte) gets. For the two bulk synchronous benchmarks, the performance gap is less a function of any inherent difference between the MPI bandwidth and that of the lower-level API (the two tend to be similar on each system), than of the cost of providing the illusion of one-sided communication under MPI 1.1.

Currently, our GASNet-MPI implementation works by having each host perform a set of nonblocking receive calls in advance, with varying buffer sizes, so that a buffer of the correct size is generally immediately available for any incoming message. This allows a remote processor to complete a message send before the target processor may even be aware that the message has arrived. This buffering strategy requires that the message be copied twice, however (once on the source node, to add header specifying the destination address, and once at the target node, to move the data from the buffer to the final destination).

The GASNet-MPI implementation could probably provide better performance for large messages if it used the initial receive buffer to inform the target CPU to set up a receive call that placed the data directly into the destination memory, thus avoiding buffering. However, this would still be inherently slower than the elan-based direct RDMA version: the transaction would incur an additional rendezvous (in the form of a set of MPI messages) to set up the direct receive call (and likely an additional network round trip as MPI did its own internal rendezvous), followed by the direct RMA transfer, and then a final MPI message from the destination processor informing the initiator that the message had completed (MPI does not provide built in notification of remote completion, but it is required for GAS language semantics, which sometimes need to guarantee target completion for remote writes). The transaction would also need to wait initially for the remote processor to poll the network to see the initial setup message. For messages of a sufficiently large size, however, these costs ought to be amortized to the point where the MPI layer's peak bandwidth performance might asymptotically approach that of the elan layer (when the remote node is well-attentive to the network). It unclear what the crossover message size would be for this sort of algorithm, however, and it would need to be tuned for each different type of network (and possibly on a per-machine basis), thus lessening the portability benefit of using MPI.

The performance of the naïve CG benchmark (which uses an average message size of only 8 bytes) is less ambiguous. For applications which use small messages, elan provides a significant speedup, allowing the application to run more than four times faster. Clearly the communication performance obtained by directly targeting the native network API is more suitable than a solution layered on MPI for supporting an incremental programming model for parallel applications and for inherently fine-grained applications.

It is unclear if MPI 1.1 will ever be able to provide GAS languages with the same performance as native network APIs for small and medium-sized one-sided messages. The popularity of MPI has meant that vendors are willing to expend considerable effort tuning their implementations, and certain trends are promising, such as the offloading of some of MPI's message-passing logic onto dedicated network co-processors. Although such co-processors are not likely to improve end-to-end message latency, they might lower the host CPU's software overheads to levels comparable to that of lower-level network APIs, which might in turn allow a UPC implementation to hide the latency by performing unrelated work in the interim.

3 MPI-RMA as a compilation target for GAS languages

The most recent version of the MPI specification (MPI 2.0) extends MPI 1.1 with a number of new features. In particular, Chapter 6 of the specification adds support for "one-sided communications" (also called "Remote Memory Access (RMA)"). This extension supplements the traditional two-sided MPI communication model with a one-sided interface that can take advantage of RMA network hardware, with its shared memory paradigm, and lower latency and software overhead costs.

On the surface MPI-RMA thus seems like a natural fit for GAS languages. However, the rest of this document explains why MPI-RMA does not meet the requirements laid out in section 1.1 for implementing global address space languages. Essentially, the strong restrictions placed on memory access patterns by the API and the weakness of its semantic guarantees make it unusable for GAS language implementation purposes.

Note that we are interested in writing portable code, and are therefore not concerned with the behavior of particular implementations of MPI-RMA (which may happen to relax some of these constraints, and/or have well-defined semantics for conditions the specification labels as erroneous), but rather with the guarantees provided by *any* MPI-RMA implementation (including one which aggressively exploits the intentionally under-specified aspects of the API).

3.1 Basics of the MPI-RMA API

The semantics of MPI-RMA are fairly complex. Here we present only an overview of its usage, then proceed to discuss those aspects of the API that affect GAS language implementations. The reader may consult the MPI 2.0 specification [24] for details.

The MPI-RMA API revolves around the use of abstract objects called "windows" which intuitively specify regions of a process's memory which have been made available for remote operations by other MPI processes. Windows are created using a collective operation (MPI_Win_create) called by all processes within a "communicator" (a group of processes in MPI terminology), which specifies a base address and length (which may be different on each process), and is permitted to span very large areas (e.g., the entire virtual address space). All three one-sided RMA operations (MPI_Put, MPI_Get, MPI_Accumulate) take a reference to such a window, an offset into the window, and a rank integer to indicate which process is the remote target. All one-sided operations are implicitly non-blocking and must be synchronized using one of the synchronization methods described below.

3.1.1 Active Target vs. Passive Target

There are 2 primary "modes" in which the one-sided API can be used, named "active target" and "passive target". The primary semantic distinction is whether or not cooperation is required by the remote node in order to complete a remote memory access. All RMA operations on a window must take place within a synchronization "epoch" (with a start and end point defined by explicit synchronization calls), and operations are not guaranteed to be complete until the end of such an epoch. The active and passive target modes differ in which process makes these synchronization calls.

Active target operation requires synchronization functions to be called on both the origin process (the one making the RMA get/put accesses) and the target process (the one hosting the memory in the referenced window). The origin process calls MPI_Win_start/MPI_Win_complete to begin/end the synchronization epoch, and the target process must cooperate by calling MPI_Win_post/MPI_Win_wait to acknowledge the beginning/end of the epoch (there is also a collective MPI_Win_fence operation which can be substituted for one or more of these calls). In any case, this required cooperation effectively destroys the possibility of implementing the truly one-sided operations that we wish to provide in GAS languages using active-target mode RMA.

Passive target operation provides more lenient synchronization. In passive-target operation, only the originating process calls synchronization functions (MPI_Win_lock/MPI_Win_unlock) to start/end the access epoch. As with active target, all RMA accesses must take place within such an epoch and are not guaranteed to complete until the MPI_Win_unlock call completes. There are two forms of MPI_Win_lock–shared and exclusive. MPI_Win_lock(exclusive) enforces mutual exclusion on the window and the RMA operations performed within the epoch–i.e., it conceptually blocks until it can start an exclusive access epoch to the window, and no other processes may enter a shared or exclusive access epoch for that window until the process with exclusive access unlocks (the semantics are actually slightly weaker than this, but the intuition is correct). MPI_Win_lock(shared) allows other concurrent shared epochs from other processes. The spec recommends the use of exclusive epochs when executing any local or RMA update operations on the memory encompassed by the window to ensure well-defined semantics (section 6.4.3, p.131).

3.2 Restrictions on the Use of Passive Target RMA

The interface described thus far for passive target RMA seems reasonable, however unfortunately there are a large number of restrictions on how it may be legally used. Here are some of the most important restrictions:

1. Window creation is a collective operation–all processes which intend to use a window for RMA (including all intended origin and target processes) must participate in the creation of that window (section 6.2.1, p.110).

2. Implementors may restrict the use of passive-target RMA operations to only work on memory allocated using the "special" memory allocator MPI_Alloc_mem (section 6.4.3, p. 131). This prevents the use of passive-target RMA on static data and forces all globally-visible objects to be allocated using this "special" allocation call (no guarantees are made about how much memory can be allocated using this call, and some implementations may restrict it to a small number of pinnable pages).

3. It is erroneous to have concurrent conflicting RMA get/put (or local load/store) accesses to the same memory location (section 6.3, p.113).

4. The memory spanned by a window may not concurrently be updated by a remote RMA operation and a local store operation (i.e., within a single access epoch), even if these two updates access different (i.e., non-overlapping) locations in the window (section 6.3, p.113).

5. Multiple windows are permitted to include overlapping memory regions, however it is erroneous to use concurrent operations to distinct overlapping windows (section 6.2.1, p. 111).

6. RMA operations on a given window are only permitted to access the memory of single process during an access epoch (section 6.4.3, p.131)

3.3 Implications of the MPI 2.0 semantics

We now investigate the implications of the above restrictions on our effort to implement remote accesses in a GAS language.

Restriction #1 implies that a GAS language implementation can not use a separate window per shared object, because that would require a collective operation for the allocation of shared objects, and object allocation in GAS languages is often required to be a purely non-collective, local operation. This means many or all shared objects would need to be coalesced within a single window. But while arbitrarily large regions (like the entire virtual address space) can be mapped within a single window, this would severely limit concurrency due to restriction #4. The same shared object can be mapped into several windows, but due to restriction #5 this probably is not useful.

Restriction #2 implies that all potentially shared objects must be allocated using MPI_Alloc_mem(). This restriction alone may be enough to make MPI-RMA unsuitable for many GAS languages, unless MPI implementations permit large amounts of memory to be allocated using this function (recall that a significant fraction of all data allocated by GAS programs is typically accessed remotely at some point in program execution).

Restriction #6 means that an implementation would probably need at least one separate window for each target process, as otherwise RMA operations from one process to different target processes will be unnecessarily serialized. This may present scalability problems for large numbers of nodes, depending on how windows are implemented.

The underlying concept addressed by restriction #3 is fundamental to shared-memory programming and nothing new. GAS languages generally specify that conflicting accesses to a single memory location will store an undefined result to the location (for conflicting writes) or return an undefined value (for the read in a read-write conflict). However, the MPI restriction is unfortunately much stronger–it says such conflicting accesses are *erroneous*, which implies that any resulting behavior after such a violation is possible (e.g., the MPI implementation would be within its rights to consider this a fatal error). Unfortunately, it is not feasible to statically detect all such conflicting accesses (from different processes) in user-provided code without application-specific information (or perhaps even with it). Using a great deal of compiler analysis, a conservative superset of the conflicting accesses could be generated, but in a weakly typed language such as UPC, this is likely to include most of the accesses. It is impossible to detect such conflicts at runtime without global communication. In truth, the compiler analysis required simply to decide that it is safe for a single process to include any other RMA accesses within the same epoch as an RMA put or accumulate is non-trivial, although this problem is an issue of sequential alias analysis which is considerably better understood. MPI_Win_lock(exclusive) can be used to conservatively prevent concurrent conflicting accesses from distinct processes (by wrapping every RMA Put within its own exclusive epoch). However, because this locks the entire window and serializes all access epochs to that window, this would drastically reduce the concurrency of accesses to distinct (i.e., non-conflicting) memory locations that happen to reside within the same window (which would be very bad if the entire shared memory resided in a single window). This also effectively nullifies the ability to perform non-blocking puts. Another option which might help is to replace all RMA puts with RMA accumulate operations where the reduction operation is "MPI_REPLACE"– this has the same semantics as an RMA put, but conflicting accumulate operations have well-defined semantics (they behave as if the conflicting accumulates happened in some serial order)–however, conflicting RMA gets and local loads to the same data would still be erroneous within the access epoch, so an untenable amount of conflict analysis would still be required.

Restriction #4 is particularly onerous for GAS language implementations. It prevents the local process from making any changes to memory that lies within a window during a remote access epoch to that window, even to different (i.e., non-conflicting) memory locations. This implies some form of synchronization between the origin and target process when accessing these locations, which implies truly one-sided communication is not possible. The MPI spec recommends the local process perform all updates to the local memory which falls within a window inside an exclusive epoch on that window. Because every access to a language-level local pointer in UPC and Titanium is potentially an access to a shared memory location residing locally, in the absence of other information *every* local store operation would need to be wrapped within an exclusive access epoch (possibly prefixed with a check of whether the accessed location resides in a window). Similarly, every local load which could potentially conflict with a remote RMA put or accumulate to that location would need to be wrapped within a shared access epoch. The performance implications of adding such overheads to what should be simple local memory load/store instructions are staggering. Note that it is not sufficient for the local process to simply maintain a permanent epoch on its own window, because this would prevent remote RMA operations on that window from making progress. Finally, restriction #4 also implies that the entire virtual address space of a process can not be included in a window, because this would include private memory that is constantly changing, such as the program stack.

The combination of these effects makes the MPI-RMA effectively unusable by GAS language implementations: it is extremely unlikely that a layer could be written on top of MPI-RMA that would provide an efficient implementation of GAS language communication semantics. Given that adherence to some of the MPI-RMA restrictions involve such difficult compiler problems as conflict and alias analysis, it it likely that many other parallel languages would also find MPI-RMA a difficult interface to target. It seems likely that only languages that effectively expose most of MPI-RMA's restrictions and programming disciplines to the user at the language level would find MPI-RMA a useful compilation target.

4 Related Work

A number of papers on MPI-RMA performance are available. Luecke and Hu [20] evaluated MPI-RMA performance on the Cray SV1, and Luecke et al. [21] compared MPI-RMA performance to that of the vendor-supplied SHMEM libraries on the Cray T3E and SGI Origin 2000 architectures, finding that the SHMEM interface significantly outperformed the MPI-RMA implementation available at the time. Traff et al. [29] examined the performance of MPI-RMA on NEC SX-5 system, and found its data transfer performance to be similar to that of the MPI 1.0 two-sided API. Matthey and Hansen [22] report that a production molecular dynamics simulation code ran 10-70% faster on an Origin 2000 system after being converted from MPI-1 to MPI-RMA.

Our strategy of implementing one-sided, asynchronous messaging on top of MPI 1.1 is not a new one. Dobbelaere [9] reports implementing a custom one-sided communication layer over MPI 1, and Booth and Mourao [6] discuss implementating the MPI-RMA API itself over MPI 1.1.

It appears from Smith [28] that the MPI-RMA pioneers decided early on to require a separate set of allocation functions to allocate data that can be accessed by passive-target one-sided operations. For an alternative approach that allows the entire virtual memory space to be accessed by RDMA (even for the tricky case of pinning-based networks such as Myrinet), see the description of lazy registration/pinning methods in [3].

Gropp [13] reviews some of the reasons for MPI's enduring success in the parallel computing community.

5 Conclusion

We have shown that despite having certain desirable characteristics, neither the MPI 1.1 nor the more recent MPI 2.0 RMA extensions make an attractive compilation target for Global Address Space languages. MPI 1.1, despite its wide availability and often highly tuned performance, imposes too many overheads with its two-sided messaging paradigm for GAS languages, particularly those in which small message performance is important. The newer MPI-RMA API imposes too many semantic restrictions to be a useful portable compilation target, at least for parallel languages which allow aliasing, data conflicts, and/or the illusion of a single, arbitrarily accessible shared address space.

The fact that MPI-RMA makes a poor compilation target for at least certain classes of parallel languages does not mean that the API is not useful to application programmers, who are capable of globally structuring their application to abide by the specification's semantic restrictions. The purpose of this paper, in short, is not to malign MPI-RMA, but rather to explain why it is not suitable as a portable communication layer for implementing parallel languages such as Titanium, UPC and Co-Array Fortran. We sincerely hope that the MPI-RMA interface can be revised in future versions of the MPI specification to relax some of the problematic semantic restrictions described in this paper. Perhaps with some careful changes made to accommodate the requirements of parallel high-performance language implementors, it would become more useful as a portable compilation target for these languages.

References

[1] AMMPI home page. http://www.cs.berkeley.edu/~bonachea/ammpi.

[2] D. Bailey, E. Barszc, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. The NAS parallel benchmarks. Tech Report RNR-94-007, RNR, March 1994. http://www.nas.nasa.gov/Software/NPB/.

[3] C. Bell and D. Bonachea. A new DMA registration strategy for pinning-based high performance networks. In *Workshop Communication Architecture for Clusters (CAC03) of IPDPS'03, Nice, France*, 2002.

[4] C. Bell, D. Bonachea, Y. Cote, J. Duell, P. Hargrove, P. Husbands, C. Iancu, M. Welcome, and K. Yelick. An evaluation of current high-performance networks. In *IPDPS 2003*, 2003. http://upc.lbl.gov.

[5] D. Bonachea. GASNet specification, v1.1. Tech Report UCB/CSD-02-1207, U.C. Berkeley, October 2002. http://www.cs.berkeley.edu/~bonachea/gasnet.

[6] S. Booth and F. E. Mourao. Single sided MPI implementations for SUN MPI. In *Supercomputing*, 2000.

[7] W. Chen, D. Bonachea, J. Duell, P. Husbands, C. Iancu, and K. Yelick. A performance analysis of the Berkeley UPC compiler. In *17th Annual International Conference on Supercomputing (ICS)*, 2003. http://upc.lbl.gov.

[8] Compaq UPC compiler. http://www.tru64unix.compaq.com/upc/.

[9] J. Dobbelaere and N. Chrisochoides. One-sided communication over MPI-1. http://citeseer.nj.nec.com/dobbelaere01onesided.html.

[10] T. A. El-Ghazawi, W. W. Carlson, and J. M. Draper. UPC specification, v1.1, March 2003. http://upc.gwu.edu.

[11] Elan programmer's manual. http://www.quadrics.com.

[12] GM reference manual. http://www.myri.com/scs/GM/doc/refman.pdf.

[13] W. D. Gropp. Learning from the success of MPI. In *International Conference on High Performance Computing (HiPC 2001)*, August 2001.

[14] P. Hilfinger, D. Bonachea, D. Gay, S. Graham, B. Liblit, G. Pike, and K. Yelick. Titanium language reference manual. Tech Report UCB/CSD-01-1163, U.C. Berkeley, November 2001.

[15] Infiniband trade association home page. http://www.infinibandta.org.

[16] A. Krishnamurthy and K. Yelick. Analyses and optimizations for shared address space programs. *Journal of Parallel and Distributed Computing*, 38(2):130–144, 1996.

[17] Using the LAPI. http://www.research.ibm.com/actc/Opt_Lib/LAPI_Using.htm.

[18] B. Liblit and A. Aiken. Type systems for distributed data structures. In *Conference Record of POPL '00: The 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Boston, Massachusetts, 19–21 Jan. 2000.

[19] B. Liblit, A. Aiken, and K. Yelick. Type systems for distributed data sharing. In *SAS '03: The 10th International Static Analysis Symposium*, Lecture Notes in Computer Science, San Diego, California, June 11–13 2003. Springer-Verlag.

[20] G. R. Luecke and W. Hu. Evaluating the performance of MPI-2 one-sided routines on a Cray SV1. Technical report, December 21, 2002.

[21] G. R. Luecke, S. Spanoyannis, and M. Kraeva. The performance and scalability of SHMEM and MPI-2 one-sided routines on a SGI Origin 2000 and a Cray T3E-600. In *PEMCS*, December 2002.

[22] T. Matthey and J. Hansen. Evaluation of MPI's one-sided communication mechanism for short-range molecular dynamics on the Origin 2000. In *Proceedings of PARA2000, The Fifth International Workshop on Applied Parallel Computing*, 2000.

[23] S. P. Midkiff and D. A. Padua. Issues in the optimization of parallel programs. In *Proceedings of the 1990 International Conference on Parallel Processing*, 1990.

[24] MPI Forum. MPI–2: a message-passing interface standard. *International Journal of High Performance Computing Applications*, 12:1–299, 1998. http://www.mpi-forum.org/docs/mpi-20.ps.

[25] MPI Forum. MPI: A message-passing interface standard, v1.1. Technical report, University of Tennessee, Knoxville, June 12, 1995. http://www.mpi-forum.org/docs/mpi-11.ps.

[26] R. Numrich and J. Reid. Co-Array Fortran for parallel programming. In *ACM Fortran Forum 17, 2, 1-31.*, 1998.

[27] D. Shasha and M. Snir. Efficient and correct execution of parallel programs that share memory. In *ACM Transactions on Programming Languages and Systems, 10(2):282–312*, April 1988.

[28] A. G. Smith. Using MPI 2 one sided communications on Cray T3D. Tech report, EPCC: The University of Edinburgh, December 1995. http://citeseer.nj.nec.com/88176.html.

[29] J. Traff, H. Ritzdorf, and R. Hempel. The implementation of MPI–2 one-sided communication for the NEC SX. In *Proceedings of Supercomputing*, 2000.

[30] K. Yelick, L. Semenzato, G. Pike, et al. Titanium: A high-performance Java dialect. ACM 1998 Workshop on Java for High-Performance Network Computing, February 1998. http://titanium.cs.berkeley.edu.